Scaling to Thousands of Processors with Buffered Coscheduling*

Fabrizio Petrini

CCS-3 Modeling, Algorithms, & Informatics
Computer & Computational Sciences Division
Los Alamos National Laboratory
fabrizio@lanl.gov
http://www.c3.lanl.gov/~fabrizio

Abstract

In this paper we describe Buffered Coscheduling, a new approach to design the system software of large scale parallel computers. A buffered coscheduled system can tolerate inefficient programs, programs that have communication and load imbalance, and can substantially increase the resource utilization, by overlapping computation, communication and I/O over several jobs. In addition to that, Buffered Coscheduling creates a framework to easily implement fault-tolerance, arguably the most important problem to solve to build usable, large scale parallel computers.

1 Three Important Dimensions in Parallel Computing

Clusters of workstation are becoming widespread, thanks to the availability of commodity components that can be integrated in a single machine. For example, by using blade servers, it is now possible to stack several hundreds of processors in a single rack. It is easy to predict that in the near future large scale clusters, with thousands of processors, will move out of the boundaries of the research labs and will find their way into commercial computing. In order to have a critic view of the current state of the art in parallel computing we will organize our discussion along three important dimensions (see Figure 1).

- Programming and Performance
- Resource Utilization and
- Fault-Tolerance.

*The work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36

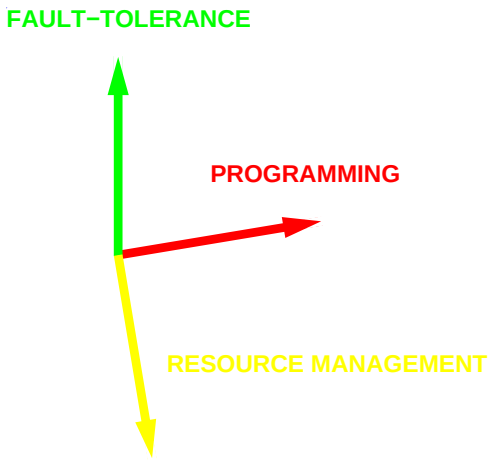

Figure 1: Three important dimensions in parallel computing

1.1 Programming and Performance

When we talk about a parallel computer, application performance is probably the main concern. In fact, the traditional goal of a parallel computer is to solve a given problem in as little time as possible. But experience has shown that extracting the full performance from a parallel computer is not an easy task.

Most parallel programs are coded using MPI. When writing a program in MPI, the user must pay attention to a large number of details. It must divide the input data set in a number of distinct domains, map these logical domains onto a set of processes, define an inter-process communication schedule, typically based on pairs of matching sends and receives, set synchronization points, etc. This process is tedious and error-prone, and it is usually difficult to debug a parallel program and prove its correctness. Obtaining good performance from a working MPI program can also be a challenging task. A load imbalance in a single process can slow down the whole parallel program. The communication schedule is also critical to achieve good performance. Many parallel programs have a self-synchronizing behavior, often called bulk-synchronous, and a slight load or communication imbalance can severely impact the run time of the whole program. Simply put, parallel programs are “unstable” from a performance point of view, and minor problems can have a severe impact on the overall performance.

MPI is a greatest common denominator that guarantees portability across a wide range of parallel architectures. But, in general, there is little guarantee that the same level of performance of a program tuned for a particular architecture will be maintained when moving to another architecture. In sequential computing, we take for granted that an existing program will take advantage almost automatically of technological improvements (faster processors, memory, I/O bus etc), by simply recompiling the source code. This still doesn’t happen with parallel computers. For this reason, large investments for developing parallel software are still limited, and in most cases confined to the national labs and a few companies. Developing parallel code is clearly a major obstacle to the widespread use of parallel computers. The typical cost of a ASCI-class physical simulation code is in the order of tens of millions of dollars and it is probably

as expensive as the hardware on which it runs. But these codes have a life expectancy of several decades, while the hardware becomes outdated in just a few years.

The development of scalable communication libraries and system software is also very time consuming. For example, the experience with the largest ASCI machines as Blue Mountain and the 30T machine at Los Alamos, or ASCI White at Livermore, shows that it takes a few years to eliminate all the bugs (typically race conditions) and scalability bottlenecks. It may sound odd, but usually these machines become fully functional shortly before they are phased out. This is influenced by the complexity of communication libraries as MPI, which have more than 200 function calls, and by the inherent non-determinism of the communication layer.

1.2 Resource Management

Most large scale computers allocate the resources in space sharing. For example, when we launch a parallel job, the resource management system in use on the machine allocates a subset of processors/processing nodes, all their memory, network and I/O subsystems. This winner-take-all approach has several limitations. Some in-depth studies and characterizations of large scale ASCI codes show that these programs have a pronounced bulk-synchronous behavior. All the processes compute the cells in a given domain all together for a few milliseconds. Then they all stop and perform and exchange of data, typically according a well-defined communication stencil with a set of logical neighbors, again for a few milliseconds. Less frequently, they perform I/O. The typical I/O patterns are checkpointing (roughly 80% of I/O traffic) and data collection (the remaining 20%), in order to analyze the progress of the computation. In all these phases, only one type of resource is used. For example, when the job is computing, both network and I/O are idle, when it is communicating, all processors and I/O are not in use, and when it is doing I/O only a small fraction of the network is used.

If we give a critical look to how resources are allocated in a parallel computer, we can find a strong resemblance between the state of the art and the operating systems for PCs of the early eighties, as Microsoft DOS. In such systems, the user was able to run a single program at a time. Any PC user, would nowadays find unacceptable to run a single program at a time (for example reading e-mail, using a web browser, accessing the file system, etc) and we take for granted that the operating system can multitask several activities at the same time. We can print a file, and while doing that, we can continue using the available resources for executing other tasks. Unfortunately, this is not yet possible in a parallel computer.

Resource managers have also many scalability limitations. The typical resource manager is structured as a collection of deamons, each one running on a distinct node, that communicate with slow, point-to-point TCP/IP connections. The existing production systems show that this design is inherently not scalable, due to both the overhead of the communication library and to the algorithmic design of the communication patterns that distribute and gather control messages and data.

1.3 Fault-Tolerance

The last, but probably most important dimension is the fault-tolerance. An ASCI-class machine as the 30T, will have more than 12000 processors, more than 6000 network interfaces, tens of thousands of cables. A preliminary analytical evaluation of the MTBF

shows that the machine will be up, on the average, only a few hours. The probability that a 4096 processor job, running for 5 hours will successfully complete is less than 50%. A common approach to alleviate this problem is to partition the machine in smaller segments, and use these segments as independent machines. Inside each segment, the applications run a user-level checkpointing algorithm at regular intervals. The checkpoint is implemented with a rather simple algorithm: the whole application is halted, and each process dump into the file system the relevant state of the computation. The checkpoint I/O traffic is roughly estimated as 80% of the whole I/O traffic.

An alternative approach is to checkpoint the status of a job while it is running, in an incremental and transparent way. Unfortunately, the state of the art in dynamic checkpointing is still in its infancy. The main algorithmic challenge behind dynamic checkpointing is to identify a consistent global state, to which the job can rollback, should a fault occur. The identification of such global state is not trivial, because the algorithms must take into account many details, such as the messages in transit and the interaction with the file system, and because these algorithms can only rely on limited, local information on the global state. If the distributed algorithms are not carefully designed, the only valid global state can be the initial one at launch time, thus generating the so called "domino effect". The state of the art in dynamic checkpointing is Egida, a transparent, low-overhead environment that incrementally checkpoints the status of an MPI job at run time, developped at UT Austin. The initial results show the scalability of this approach is limited to a few nodes, and it is not clear whether it will be possible to scale to a large number of nodes using only local information, in the next few years.

2 A case Study: 3-D Simulation of a Nuclear Weapon

Scientists at Los Alamos and Livermore national laboratories have recently completed two of the largest computer simulations ever attempted, the first full-system three-dimensional simulation of a nuclear weapon[1]. The Los Alamos simulation used more than 480 million cells on 1,920 of the 8,192 processors on the ASCI White machine at Livermore. The actual processing time was 2,931 wall-clock hours or 122.5 days – more than 6.6 million CPU hours. The Livermore simulation ran on more than 1024 processors of the same machine and took 39 days to execute.

An in-depth performance evaluation of Sage [4], a sanitized version of the Los Alamos simulation which maintains the same properties of the original code (computational granularity, communication pattern, data set etc), shows that, when we run this application on a large number of processor, no more than 50% of the time is spent computing and the remaining time is spent communicating. What is more important, this application display a bulk-synchronous behavior, that strictly alternates these two phases. So, when the processors are computing there is no communication in the background and vice versa. This behavior is more pronounced when we use a larger number of processors.

In addition to that, about 5 minutes every hour are spent doing a user-level checkpoint. It is interesting to note that these applications would require all the processors, all the memory and all the time, if possible. But there is a practical limit to the resources that can be effectively allocated. In fact in a large configuration the expected

[1] http://www.lanl.gov/orgs/pa/newsbulletin/2002/03/08/text01.shtml

MTBF is only few hours, so a checkpoint should take place very frequently, up to a point where very little useful work gets done. The frequency of the checkpoint is empirically determined by considering the perceived number of faults per unit of time and the overhead to perform the checkpoint itself.

We can thus draw two important considerations.

1. Due to the lack of fault-tolerance, large scale machines cannot be fully used as capability engines.

2. If we sum the idles times due to communication and checkpoint, we can see that we have the resources necessary to implement another (virtual) supercomputer, using the same hardware, which has the same computation capability.

3 Buffered Coscheduling

Buffered Co-Scheduling (BCS) is a new methodology that can substantially increase resource utilization, simplify the development of parallel code by tolerating inefficient programs and enhance fault-tolerance in a large scale parallel computer [1].

BCS multitasks parallel jobs. That is, instead of overlapping computation with communication and I/O within a *single parallel program*, all the communication and I/O which arises from a *set of parallel programs* can be overlapped with the computations in those programs. We propose a new approach based on strobing heart beats at regular intervals, or time-slices, to tightly synchronize the processors and to schedule the communication and the computation. To implement this multitasking, buffered coscheduling relies on two techniques. First, the communication generated by each process is buffered and performed at the end of regular intervals (or time-slices). By delaying communication, we allow for the global scheduling of the communication pattern. Second, a strobing mechanism performs an exchange of control information at the end of each time-slice. The goal is to move away from isolated scheduling algorithms (where processors make decisions based solely on their local status and a limited view of the remote status) to more outward-looking or global scheduling algorithms.

3.1 Communication Buffering

In BCS every communication primitive is implemented by filling in a descriptor with all the important information. For example, when executing a point to point send, the descriptor contains information on the source and destination processes, communication buffers, tag matching etc. If the communication primitive is blocking, the process is suspended and will be waken up when all the communication protocol has been successfully completed, for example by a local scheduler or by the OS, depending on the type of implementation. When executing a non blocking primitive, the process is not interrupted. In both blocking and non-blocking case, the actual communication protocol is not eagerly performed, as done in virtually all existing implementations of communication libraries. The goal is to collect as much information as possible on the global state of the machine, before sending a message. Also, by delaying the communication, we greatly simplify the implementation of the fault-tolerance and the

communication library itself, as explained below. Depending on the type of implementation, the actual communication protocol can be run by a daemon process, by the kernel or, more aggressively, by a network interface, using a low-overhead protocol.

3.2 Strobing

At the core of BCS there is the strobing algorithm, which tries to strictly schedule all the activities in a parallel machine at regular intervals, called time-slices. Some important steps of this algorithm are listed below.

1. Distribution of the start strobe signal: this control message is distributed to all processing nodes, possibly with little skew. Strobe signals are delivered at every time-slice. Depending on the architectural support available, the start strobe can be delivered using a dedicated network, as the control network available in the Connection Machine CM-5 [5], through some hardware multicast capability of the data network, if available, of emulated with a tree-based software multicast.

2. Distribution of control information: after the delivery of the start strobe, all the nodes distribute the relevant control information, for example the communication buffers, in order to schedule all the communication.

3. Perform the actual scheduling of the communication. The main goal is to send in each time-slice only the messages that can be actually be delivered by the network and sent/received by the processing nodes in the current time-slice. For example if a processing node can deliver β MB/sec, during a time-slice of length Δ sec, it will only be able to send/receive $\beta\Delta$ MB of data, in the optimal case. So there is no advantage in trying to overcome this limit. Also, the scheduling algorithm can delay the transmission of messages whose destination buffers are paged out, thus avoiding expensive re-transmissions.

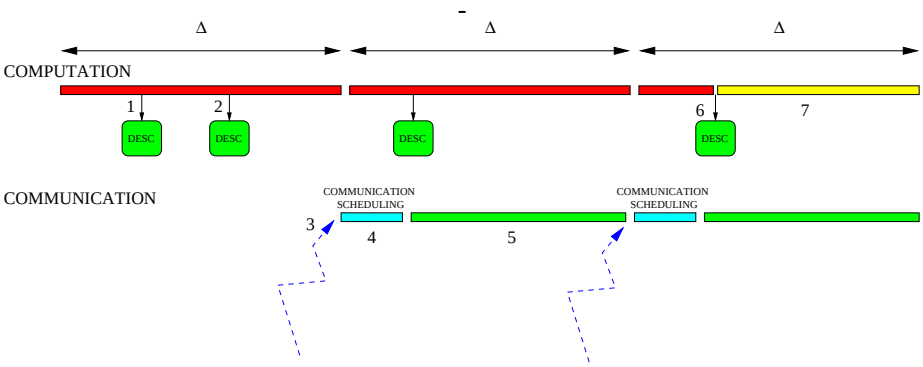

Figure 2: Strobing Algorithm

Some intuition on the strobing algorithm is provided in Figure 2. The Figure describes a possible execution scenario on a specific processor. The processor executes a process which issues two non-blocking calls (Figure 2 step 1 and 2) within the first time-slice Δ. The calls fill in two communication descriptors and the process continues its computation. After the arrival of the start strobe (step 3), we try to schedule the

communication by matching the pending descriptors. For example, we try to resolve any pairs of matching sends and receives. At this point we schedule the subset of communications that can be actually delivered in the current time-slice. We try to overlap those communications with the ongoing computation. In the presence of a blocking call (step 6), the calling process is suspended and another process belonging to another job that is ready to run is scheduled on the processor. It is worth noting that the strobing time-slice is different from the job scheduling time-slice which, in the general case, is an integer multiple of the strobing one.

3.3 Tolerance to Inefficient Programs

BCS is able to tolerate two types of inefficiencies in the user applications:

1. transient load imbalance and
2. high communication overhead.

The traditional approach in parallel program development, is to reduce load imbalance within the single program. This can be very time consuming, in particular with those applications, as Adaptive Mesh Refinement (AMR) where the processor load varies at run time. Rather then leaving the burden on the programmer, and increasing the cost of parallel software, BCS tries to balance the processor load over a set of parallel jobs at run time. Figure 3 provides some intuition on how BCS can compensate transient load imbalance over multiple jobs. An extensive simulation analysis [2] shows that, if we put together a few parallel jobs with a pronounced and randomly distributed load imbalance, BCS can achieve almost optimal processor utilization. BCS in its basic form cannot handle permanent load imbalance, the one generated by applications that have some processes consistently underloaded/overloaded. This problem can be handled with a built-in process migration mechanism, discussed below.

When we scale an application to a large number of nodes, we inevitably expand the communication time. We can tolerate the communication overhead in the same way we tolerate load imbalance, by overlapping computation and communication of several jobs. Figure 3 shows how the communication generated by three jobs can be globally optimized.

BCS can thus transform a collection of ill-behaved user programs in a single, well behaved, system program. BCS can perform these optimizations *without changing the individual applications*. This is an extremely important aspect, because program development and optimization can be very expensive.

3.4 Improved Resource Utilization

Many studies in the literature show that a gang scheduled system can have a response time orders of magnitude faster than a spaced-shared one. In a recent work, we showed that jobs spend most of their time waiting, rather than running (75% vs 25%) [6]. Gang scheduling substantially reduces the first term, because jobs can enter the system much earlier, by slightly increasing the second term, because running jobs share the processors.

BCS further improves these results by overlapping the use of the resources between multiple jobs. We estimated, using the utilization logs over a period of six months of

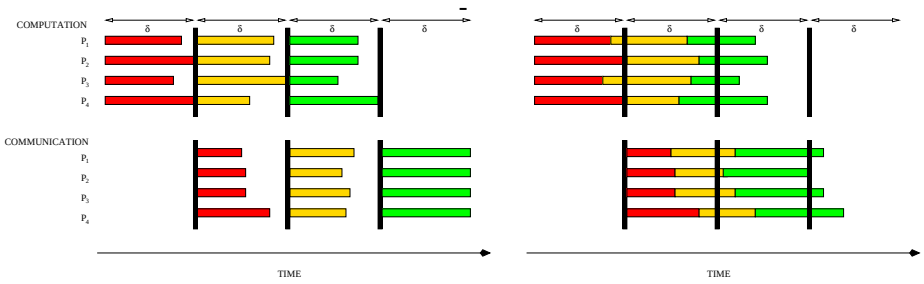

Figure 3: Filling in computation and communication holes

Nirvana, a 1 TeraOp unclassified machine at Los Alamos, that we can almost double resource utilization.

3.5 Enhanced Fault-Tolerance

BCS creates a framework to implement a fault-tolerance model based on incremental checkpoints and the use of spare nodes to replace the faulty ones over time.

BCS schedules all communication in order to have a quiescent system, possibly with no messages in transit, at the end of each time-slice. In this way we can clearly identify a valid state to checkpoint, which is simply represented by the memory image of each process plus the communication descriptors at the end of specific time-slices. In fact, it is enough to mark as "dirty" those pages that have been overwritten during each group of time-slices that represent a checkpoint quantum and flush them to safe storage. The checkpoint traffic can be scheduled as a background traffic and can be sent together with the high priority network traffic. An interesting by-product of BCS is that the same infrastructure that implements the checkpoint can be also used to implement process migration, and hence address load imbalance.

The strobing algorithm is a key point to implement fault-detection. In fact, it is possible to combine to the start strobe phase a diagnostic phase to check the status of the nodes, and identify the faulty ones, if needed.

4 Preliminary Results

In the last year we obtained a number of scientific and technical results that build a solid foundation for the success of BCS. They address several aspects, that include an in-depth performance evaluation of an important ASCI application, an innovative way to implement the strobing algorithms in few tens of microseconds in thousands of nodes, a study of the dynamics of incremental checkpoint and an initial implementation of BCS.

4.1 Performance Evaluation of Sage

In [4] we have provided an in-depth evaluation of an ASCI code, called Sage, which is considered representative of 70% of the computing cycles on ASCI machines. The analysis show that, when run in a large number of processors, these applications use

only a fraction of the computing time (about 50%) and display a pronounced bulk-synchronous behavior, which cyclically alternates phases of computation and communication.

4.2 Hardware Support for Multicast Communication

We recently proved that by using the hardware multicast of the Quadrics network [8] [7], it is possible to implement the strobing algorithm in as little as few tens of microseconds, in machine configurations with thousands of nodes. The sheer speed of the mechanism and its scalability (the run time is almost constant, irrespective the number of nodes) show that BCS can be efficiently implemented on large scale computers. The Quadrics network is currently used by some of the largest supercomputers in the world, as the Los Alamos ASCI 30T, Terascale Computer at Pittsburgh Supercomputing Center, CEA (France), LLNL etc.

4.3 Dynamic of the Incremental Checkpoint

A work in progress shows that, in most scientific applications the size of the working set that must be incrementally checkpointed is relatively small, and almost insensitive to the memory footprint of the application, but only sensitive to the size of checkpoint interval. A preliminary study of Sage, shows that on the ASCI 30T, the checkpoint-induced traffic would only require 10% of background bandwidth.

4.4 STORM: A Scalable TOol for Resource Management

We are implementing an initial prototype of BCS, called STORM [3] on Alphaserver and Intel-based Linux clusters. The experimental results show that STORM can perform the strobing algorithm in few tens of microseconds, using the hardware support for multicast provided by the Quadrics network. It can also distribute and launch a 12MB executable on a 256-processor/64-node Alphaserver cluster in less than 150 milliseconds. The gang scheduler can perform a global context switch as fast as a local scheduling decision in few hundreds of microseconds, by using an optimized algorithm running on the network interface processor. To the best of our knowledge, this is at least two orders of magnitude faster than any existing production resource managers in launching jobs, performing resource management tasks and gang scheduling.

5 Conclusion

In this paper we have provided an overview of what we think are the major problems to build usable, large scale parallel computers. Buffered Coscheduling points to a completely new direction and provides an original approach that can lead to the solution of some of those problems. We hope that our work on Buffered Coscheduling will prod insightful discussions on the problems of parallel program development, resource utilization and fault-tolerance at the “Scaling to New Heights” workshop.

References

[1] Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, volume 16, Cancun, MX, May 2000.

[2] Fabrizio Petrini and Wu-chun Feng. Scheduling with Global Information in Distributed Systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China, April 2000.

[3] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Mike Lang. Managing Large-Scale Alphaserver Clusters in the Blink of an Eye. In *CAST (Compaq User Group) 2002*, San Francisco, CA, April 2002.

[4] Darren Kerbyson, Hank Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey Wasserman, and Mike Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Supercomputing 2001*, Denver, CO, November 2001.

[5] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[6] Fabrizio Petrini and Wu chun Feng. Resource Utilization and Parallel Program Development with Buffered Coscheduling. Technical report, LOs Alamos National Laboratory, 2000. Laboratory Directed Research and Development, Exploratory Research.

[7] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[8] Fabrizio petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware-Based and Software-Based Collective Communication on the Quadrics Network. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, October 2001.